*Journal of Network and Computer Applications* (2014). 41, 182-196.
DOI: http://dx.doi.org/10.1016/j.jnca.2013.11.006

# Randomized Anagram Revisited

Sergio Pastrana[a,*], Agustin Orfila[a], Juan E. Tapiador[a], Pedro Peris-Lopez[a]

[a] *Computer Security (COSEC) Lab*
*Department of Computer Science, Universidad Carlos III de Madrid*
*Avda. Universidad 30, 28911, Leganés, Madrid, Spain*

**Abstract**

When compared to signature-based Intrusion Detection Systems (IDS), anomaly detectors present the potential advantage of detecting previously unseen attacks, which makes them an attractive solution against zero-day exploits and other attacks for which a signature is unavailable. Most anomaly detectors rely on machine learning algorithms to derive a model of normality that is later used to detect suspicious events. Such algorithms, however, are generally susceptible to evasion by means of carefully constructed attacks that are not recognized as anomalous. Different strategies to thwart evasion have been proposed over the last years, including the use of randomization to make somewhat uncertain how each packet will be processed. In this paper we analyze the strength of the randomization strategy suggested for Anagram, a well-known anomaly detector based on $n$-gram models. We show that an adversary who can interact with the system for a short period of time with inputs of his choosing will be able to recover the secret mask used to process packets. We describe and discuss an efficient algorithm to do this and report our experiences with a prototype implementation. Furthermore, we show that the specific form of randomization suggested for Anagram is a double-edged sword, as knowledge of the mask makes evasion easier than in the non-randomized case. We finally discuss a simple countermeasure to prevent our attacks.

*Keywords:* Intrusion detection systems, anomaly detection, evasion attacks, adversarial classification

## 1. Introduction

Network Intrusion Detection Systems (NIDS) constitute a primary component for securing computing infrastructures. A NIDS scans network traffic and seeks to identify evidence of ongoing attacks, intrusion attempts or violations

*Corresponding author. Tel: +34 91 624 6260, Fax: +34 91 624 9429
*Email addresses:* spastran@inf.uc3m.es (Sergio Pastrana), adiaz@inf.uc3m.es (Agustin Orfila), jestevez@inf.uc3m.es (Juan E. Tapiador), pperis@inf.uc3m.es (Pedro Peris-Lopez)

of the security policies [1]. Since they are key elements of most organizations' cyberdefense systems, NIDS often become themselves targets of attacks aimed at undermining their detection capabilities. A recent survey on this topic by Corona et al. [2] identifies six general categories of attacks against them: (a) *Evasion*, where an attack payload is carefully modified so that the NIDS would not be able to detect it; (b) *Overstimulation*, where the NIDS is fed with a large number of attack patterns to overwhelm analysts and security operators; (c) *Poisoning*, where misleading patterns are injected in the data used to train/construct the detection function; (d) *Denial-of-Service*, where the detection function is disabled or severely damaged; (e) *Response Hijacking*, where carefully constructed patterns produce incorrect alerts so as to induce a desired response; and (f) *Reverse Engineering*, where an adversary gathers information about the internals of the NIDS by stimulating it with chosen input patterns and observing the response.

Reverse engineering attacks often seek to acquire knowledge that is essential to subsequently attain other attack goals. One clear example is evasion, as the attacker generally does not possess full details about the detection function and, therefore, potential ways of evading it. For example, in anomaly-based NIDS the detection function is commonly built from a set of "normal" (and attack-free) events, such as network traffic or service requests, using machine learning algorithms [3, 4, 5, 6]. The resulting model is then used to spot anomalous activities, assuming that anything deviating from normality is suspicious. A direct consequence of this operation principle is that any network packet that fits the normality model will not raise any alarm. An advanced attacker could try to modify his original attack payload so that it blends in with the normal behavior of a network, thus evading detection. These strategies were termed Polymorphic Blending Attacks (PBA) by Fogla et al. in [7], and were also demonstrated by Kolesnikov et al. [8] to evade PAYL [9], an anomaly detector based on $n$-grams (with $n = 1$ or $n = 2$) of the payload bytes for every observed packet length.

The threat posed by evasion attacks has forced some schemes to incorporate defenses to thwart them. Nearly all schemes proposed so far rely on the idea of depriving the adversary of some critical knowledge about how the payload will be processed. This can be achieved in a number of ways. For example, McPAD [10] generates various different models of normality, each one based on a distinct set of features, and uses all (or some) of them to seek anomalies. In doing so, it forces the adversary to craft a payload that looks normal to all models. A similar idea was explored by Biggio et al. in [11] by using multiple classifiers and randomly assigning weights to each of them in the final decision. Other detectors such as KIDS [12] draw some inspiration from cryptography and propose a scheme where the normality model depends upon some secret material (the key). In KIDS the key determines how the classification features are extracted from the payload. The security argument here is simple: even though the learning and detection algorithms are public, an adversary who is not in possession of the key will not know exactly how a request will be processed and, consequently, will not be able to design attacks that thwart detection.

While most such schemes successfully provide a rationale of their strength against evasion, almost none of them include in their security analysis arguments about an adversary who first reverse engineers the detection function and then uses the acquired knowledge to evade detection. Thus, for example, in PAYL an adversary can try to infer, either completely or approximately, which specific $n$-grams are not recognized as anomalous and then modify the attack (e.g., by adding carefully constructed padding as in [8]) so that it matches one of those $n$-grams. One debatable issue about reverse engineering attacks is precisely their viability and/or practical relevance. To begin with, it is assumed that the attacker can somehow observe the responses induced by inputs of his choosing, and also that he can query the NIDS with a potentially large number of payloads. Shedding doubts about the feasibility of doing this is reasonable, but from a security perspective it would be unsafe to assume that it is not possible, even if it seems hard to figure out realistic scenarios where the attacker has such a capability at his disposal.

### *1.1. Contributions and organization*

In this work we focus on Anagram [13], a well-known anomaly detector that models $n$-grams (with $n > 1$) observed in normal and attack traffic. Anagram, which can be seen as an evolution of PAYL [9] to resist evasion by polymorphic mimicry attacks, also introduces a new strategy to hinder evasion called randomization. Roughly speaking, each detector uses a secret and random mask (the key) to partition packets into several (and possibly interleaved) chunks. These chunks are reordered according to the secret random mask to produce new inputs to the same normality model; the maximum anomaly score among them is assigned to the packet. Thus, in randomized Anagram the secrecy of the mask prevents an attacker from knowing where and how to modify the original attack so as each chunk will look normal.

In this paper, we analyze the strength of randomized Anagram against key-recovery attacks and the security consequences of an adversary being able to recover the secret mask. In particular:

- We discuss adversarial settings where an attacker is given the opportunity to interact with the detector by providing carefully chosen payloads and analyzing the binary response (normal/anomalous).
- We provide an efficient algorithm to recover the secret mask using a bounded amount of queries to Anagram and discuss the experimental results obtained with a prototype implementation.
- We show that knowledge of such a secret mask, even if it is just approximate, could actually make the randomized version of Anagram weaker than the non-randomized one against evasion attacks. We present an example of how an adversary may use it to carefully distribute the attack code along the payload to bypass detection.

The rest of this paper is organized as follows. In Section 2 we provide an overview of the necessary background on anomaly detectors and evasion of NIDS. Section 3 describes the evasion problem present in PAYL and the design of Anagram, including the randomized variants. Subsequently in Section 4 we introduce an algorithm to recover the key used in randomized Anagram together with the associated adversarial model. In Section 5 we discuss the experimental results obtained with a prototype implementation, including an example of cross-site scripting (XSS) attack appropriately modified to evade detection. Finally, in Section 6 we summarize our main contributions and draw conclusions.

## 2. Background

This section provides an overview of anomaly-based NIDS and reviews the state of the art in evasion attacks against them.

### *2.1. Anomaly-based NIDS*

Anomaly detectors compare monitored activity with a predefined model of normality to detect intrusions. These systems compute the model of normality by a learning process that is usually done off-line, i.e., before deployment, although recent approaches suggest the use of online training to update the model as new normal activity is observed [14]. The monitored activity can be either network flows, service requests, packet headers, data payloads, etc. During the learning process, the system analyzes a set of normal data and computes the normal model. Afterwards, any activity that does not fit in the normal model is considered a potential intrusion. Several approaches have been proposed so far to compute the model from network data [15]. Statistic-based approaches [16] define the normal model as the probabilities of appearance of certain patterns in the training data, using thresholds and basic statistical operators such as the standard deviation, mean, covariance, etc. Heuristic-based approaches automatically generate the model of normal behavior using different approaches such as machine learning algorithms [4], evolutionary systems [17] or other artificial intelligence methods [18]. Payload-based detectors analyze application-layer data to look for attacks [19, 10]. Anagram, which is the NIDS we consider in this work, extracts $n$-grams from payloads to compute the model and detect anomalies. An $n$-gram is a sequence of $n$ consecutive bytes obtained from a longer string. The use of $n$-grams has been widely explored in the intrusion detection area, although it presents some limitations too [20].

One potential problem of anomaly-based NIDS is the need to periodically retrain the model as network traffic evolves. Online training solves this problem, but also opens the door to new threats as we discuss later. Another problem is that they still present some limitations that make them useless in real world scenarios [21], including the huge amount of false positives they produce or the difficulty to faithfully compute a model of normality. As a consequence of this, few commercial systems actually use anomaly-based approaches. The

alternative are signature-based detectors, which however are unable to detect unknown attacks and are vulnerable to polymorphism [22]. In order to properly detect real-world intrusions, a combination of both techniques is necessary.

### *2.2. Attacks on NIDS*

In recent years, NIDS have become the target of attacks aiming to thwart detection in various ways. This problem was first considered by Ptacek and Newsham [23] in a 1998 technical report that focused in subverting the way in which network traffic is processed by the NIDS, i.e. the reassembly and fragmentation policy, network flow arrangement and the like. Authors showed that, regardless of the detection algorithm used by a NIDS, system designers should take care of the packet processing in order to avoid the NIDS monitoring data different to what the endpoint being protected would actually receive. Some solutions have been proposed so far. For example, active mappings [24] force NIDS to behave identically to the endpoints being protected. Another solution, given in [25], suggests to normalize network flows so as to provide a common and precise flow that generates the same data structure in the NIDS and the endpoint.

Attacks targetting the operation of the detection engine constitute a much wider area of research, with several works published during the last decade. For instance, an over-stimulation attack occurs when an attacker knows the signatures of a NIDS and forces it to generate a huge amount of alarms [26]. In such situation, the security officer analyzing the alerts would be overwhelmed and an actual attack may not be detected. Another approach is to change the appearance of a malicious payload to bypass detection while still exploiting the system. This can be easily performed for signature-based NIDS. For instance, Vigna et al. [27] proposed a framework to generate several mutations of a given exploit. Signatures should take care of any possible mutation, which is impractical in real scenarios. In the case of anomaly-based NIDS, Fogla et al. [7] introduced Polymorphic Blending Attacks (PBAs) to target PAYL [8], a former and simpler version of Anagram. We present and discuss PBAs attacks along with PAYL and Anagram in Section 3.1.

Online-based NIDS update the detection module (i.e., the signatures or the model of normality) at detection time. An adversary can inject specially-crafted noise into one of these systems to force it to, little by little, learn a malicious behavior. This is called a poisoning attack and it has been proposed both for signature-based [28, 29] and anomaly-based [30] NIDS.

Machine Learning algorithms are often used to build up the normality model of anomaly detectors. Research on Machine Learning has traditionally focused on improving the effectiveness of the solutions, without taking into account adversarial settings. However, a current area of research has begun to explore the reliability and security of Machine Learning algorithms under adversarial models [31, 21, 32, 33].

Some of the cases presented above require the adversary to know the structure and/or behavior of the NIDS. The purpose of a reverse engineering attack is precisely to acquire this knowledge. If an adversary is able to get feedback from

the NIDS to inputs of his choosing (e.g., by observing the responses), he may use the input-output pairs to infer the detection algorithm [34]. A rather common solution to make it harder for the adversary to reverse engineer the detection engine is to use a secret or random key [10, 19]. The security of this approach has not been formally proved yet, among other reasons because most designs use these keys to hinder evasion, but do not analyze resistance against reverse engineering attacks. In this work, we show that the randomization technique proposed in Anagram [19] is vulnerable to these attacks.

## 3. Anagram and evasion attacks

Anagram is a NIDS proposed by Wang et al. in 2006 [13]. It improves PAYL, a former version presented two years before by the same authors [9]. In 2005, Kolesnikov et al. [8] showed a method to evade PAYL using Polymorphic Blending Attacks [7]. This section analyzes this evasion method and presents the mechanisms present in Anagram to counteract it.

### *3.1. Evasion of PAYL*

In 2004, Wang et al. presented an anomaly-based NIDS called PAYL (Payload Anomaly Detection) [9]. For each payload length observed in the training data, PAYL obtains a normality model by storing every $n$-gram (with $n = 1$ or $n = 2$) from all available attack-free payloads. In detection mode, PAYL first extracts the $n$-grams of the packet being analyzed. Each $n$-gram that is not present in the normal model increments the anomaly score of the packet. If the final anomaly score exceeds a predefined threshold, then the packet is tagged as anomalous.

In [8], Kolesnikov et al. described an efficient method to evade PAYL. This method was further analyzed and generalized in [7], where the authors formally presented Polymorphic Blending Attacks (PBAs). This technique consists of changing the appearance of a worm in order to make it look normal as seen by the NIDS. A PBA is composed of three parts (see Figure 1):

1. The attack vector, used to exploit some vulnerability and thus penetrate the target host.
2. The attack body, which represents the core of the attack performing the malicious actions inside the victim. It is encrypted with some simple reversible substitution algorithm using a substitution table as a key.
3. The polymorphic decryptor, which includes the substitution table to decrypt the attack body and transfers the control to it.

In order to generate a PBA against a NIDS, the attacker first learns the normality model used by the NIDS, which in the case of PAYL consists of guessing the distribution of 1-grams (bytes) that normal payloads follow. The attacker then encrypts the attack body using a simple reversible substitution algorithm, where each unaccepted byte in the attack body (i.e., one that is not included in the normal model) is substituted with an accepted byte according to a particular substitution table. The objective of such a substitution is to masquerade

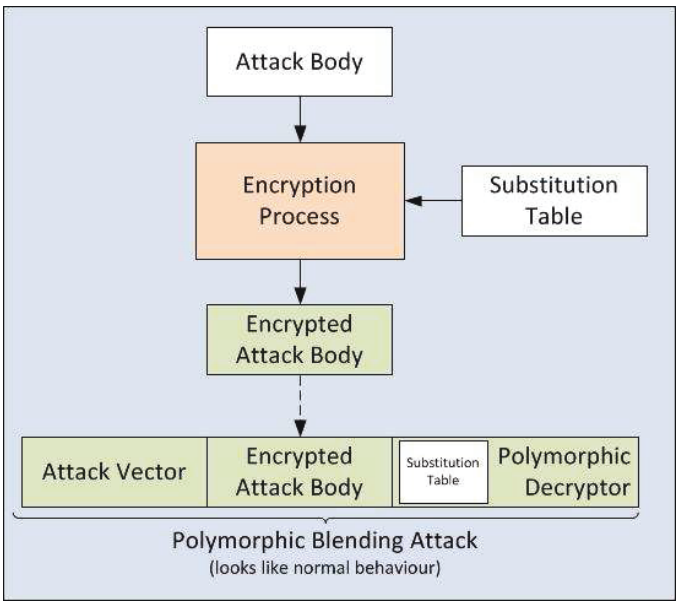



Figure 1: Scheme of a Polymorphic Blending Attack (PBA).

the attack body as normal behavior, guaranteeing that it statistically fits the normal profile. However, as the attack body is sent within the polymorphic decryptor, an optimal substitution table should itself satisfy the normal model as well. Finding such an optimal substitution table turns out to be an NP-complete problem. In [35], Fogla and Lee reduced the problem to a Satisfiability or an ILP (Integer Linear Programming) problem, which can be efficiently solved for the problem sizes involved. Finally, the polymorphic decryptor is generated. As the anomaly score of PAYL is the percentage of previously unseen $n$-grams, the PBA adds some extra normal bytes as padding to make the score lower. The attack is spread over different packets, according to the Maximum Transmission Unit (MTU) of the system, in order to make it easier for the attack to pass undetected.

### *3.2. Anagram*

In 2006, Wang et al. proposed Anagram [13] to overcome PAYL's vulnerability to PBA. Anagram uses a more complex model, and also randomizes the detection process. We next provide a brief overview of its functioning.

#### *3.2.1. Higher order n-grams*

Anagram builds a model of normal behavior by considering all the $n$-grams (for a given, fixed value of $n$) that appear in normal traffic payloads. Unlike PAYL, Anagram uses higher order $n$-grams (i.e, $n > 2$), so instead of recording single bytes or pairs of consecutive bytes, it records strings of size $n$. This obviously increments the complexity of the normal model and, therefore, requires

more computational resources. Anagram uses Bloom Filters [36] to reduce the memory needed to store the model and the time to process packets. Anagram also uses a model of bad content consisting of $n$-grams obtained from a set of Snort signatures and a pool of virus payloads. This procedure is called by the authors semi-supervised learning. In detection mode, each $n$-gram that does not appear in the normal profile increments the anomaly score by 1, except if such an $n$-gram is also present in the bad content model, in which case the anomaly score is incremented by 5. The final anomaly score of a packet is obtained by dividing the final count by the total number of $n$-grams processed. Note that the use of bad-content models makes it possible for the anomaly scores to be greater than 1. With this semi-supervised procedure, the already known attacks are taken into account, making Anagram more efficient. More details about the implementation and how Anagram works can be found in [13].

Evading PAYL requires to learn the normal profile used by the detector. This can be done stealthily by sniffing traffic destined to the victim's network from an external system. To be successful, the PBA needs to add normal $n$-grams (with $n = 1$ or $n = 2$) in order to decrease the relative frequency of unseen $n$-grams in the payload [13]. However, by using higher order $n$-grams, Anagram makes it harder to effectively reduce this frequency. The attacker has to prepare and execute the complex task of spreading the entire attack among multiple packets so as to reduce the number of unseen $n$-grams in each packet. However, this does not completely prevents Anagram from being evaded. In fact, the problem of finding a PBA against Anagram can be reduced using the techniques presented by Fogla and Lee in [35]. For this reason, Anagram does not only focus on higher order $n$-grams but also introduces the concept of randomized testing.

*3.2.2. Randomized Anagram*

A PBA always contains some number $q$ of $n$-grams that cannot be encrypted, such as the attack vector or the polymorphic decryptor. Therefore, to achieve a statistically significant percentage of valid $n$-grams, the attacker must add an extra amount $p$ of padding, with $p >> q$. In [13], a technique called *randomized testing* that aims at thwarting PBA against Anagram is described. By using such a randomization, the attacker will not know exactly how each packet will be processed and, therefore, where to put the padding to guarantee that evasion is successful.

Figure 2 graphically shows how randomized testing works, assuming that a random mask with 3 sets is used. In this case, incoming packets are partitioned into 3 chunks by applying a randomly generated mask. Such a mask consists of contiguous strings of 0s, 1s or 2s. Anagram establishes that each string must be at least 10 bits long in order to keep the $n$-gram structure of the packets (see [13] for a detailed description of the random mask generation). The mask is applied to the payload of a packet to assign each block to one of the three possible sets. Each resulting set is considered by Anagram as an independent packet formed by the concatenation of individual blocks, and are tested separately, thus obtaining different anomaly scores. The higher of these scores is the one

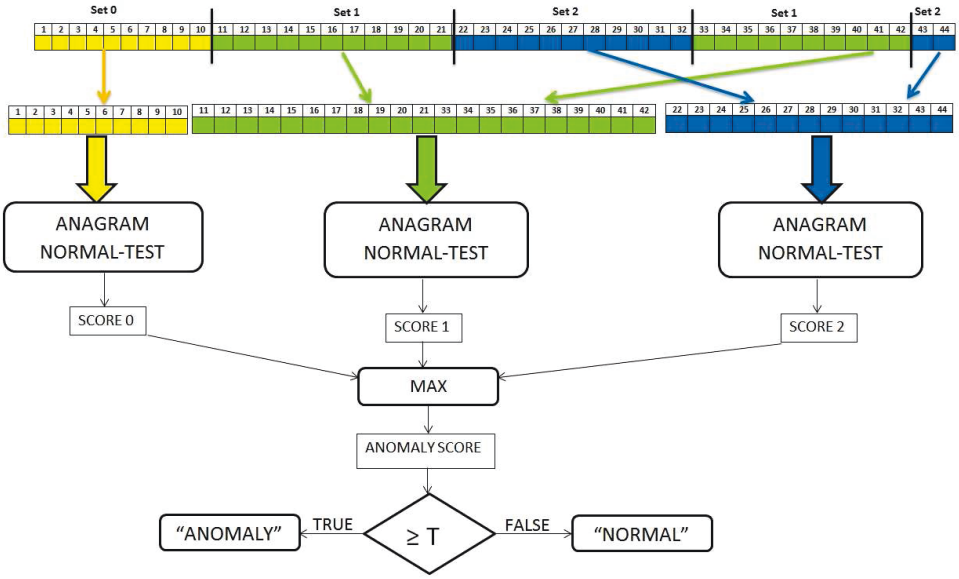



Figure 2: Computation of the anomaly score in randomized Anagram [13].

given as anomaly score of the original packet. If such an anomaly score exceeds a predefined threshold $T$, then the packet is tagged as "anomalous"; otherwise it is considered "normal."

The random mask applied in the detection process is kept secret. Consequently, an attacker does not know how the different parts of a packet will be processed in the detection process and, therefore, does not know where normal padding should be added in order to achieve an acceptable ratio of unseen $n$-grams. Thus, the first goal of an attacker pursuing to evade Anagram should be to find out the random mask used. Once this is achieved, the techniques presented in [35] to perform an ordinary PBA could be applied. Moreover, if the adversary is able to estimate the random mask used in Anagram, this can be used to easily evade the system it other ways. We describe this in detail and present an example in Section 5.4. Next section describes our algorithm to reveal the secret random mask of Anagram along with the adversarial model that we consider.

## 4. Reverse engineering attacks on randomized Anagram

In this section, we describe a reverse engineering attack against randomized Anagram. We first introduce the adversarial model required for this attack to work, including what capabilites the attacker must possess. We subsequently discuss the algorithm to recover the random mask. For simplicity, full details about the attack are provided in Appendix A.

### *4.1. Adversarial model*

In a reverse engineering attack, the attacker must possess the ability to interact with the system being attacked, often in ways that differ quite significantly from what may be regarded as normal (e.g., by providing malformed inputs or an unusually large number of them). In some cases, the ability to do so is close to the bare minimum required to learn something useful about the system's inner workings.

In the field of Secure Machine Learning, in particular when assessing the security of systems such as Anagram, one major problem comes from the absence of widely accepted adversarial models giving a precise and motivated description of the attacker's capabilities. Barreno et al. [31, 37] have recently introduced one such adversarial model and discussed various general attack categories. Our work does not fit well within this model because our main goal is not force the algorithm to misclassify an instance, but to recover one piece of secret information used during operation. Our reverse-engineering scenario is far more similar to that of Lowd and Meek [38], where the focus is on the role of active experimentation with a classifier. In this case, as emphasized in [38] it is absolutely essential for the attacker to be able to: (1) send queries to the classifier; and (2) get some feedback about properties of the query as processed by the system.

In this work, we use the adversarial model introduced in [38] to analyze the security of Anagram against reverse engineering attacks. In particular, we assume that the adversary can query Anagram with specific inputs of his choosing and analyze the corresponding responses, i.e., the adversary can:

1. Prepare a payload $p$.
2. Query Anagram with $p$.
3. Obtain the classification of $p$ as *normal* or *anomalous*.

Our emphasis in this work is on what can be attained by assuming an attacker with such capabilities. Consequently, we do not make any claims about the feasibility of the proposed attacks in real-world scenarios. However, in practical terms we identify two different settings where this adversarial model *might* materialize:

- *Offline setting.* In this case, the attacker is given full access to a trained but non-operational Anagram for a limited period of time. The attacker can freely query the system and observe the ouputs at will and without raising suspicions. For example, this situation may occur during an outsourced system auditing, in which the consultant may ask the security administrator to take full control of the NIDS for a short period of time in order to carry out some stress testing. Among the battery of tests used, he might include those queries required by the attack.

- *Online setting.* Even if the NIDS is operational, it is reasonable to assume that an attacker can send queries to the NIDS, as the ability to feed the NIDS with inputs is available to everyone who can access the service being protected. Thus for example, such queries would be arbitrarily chosen

payloads sent to an HTTP, FTP, SQL, etc. server. Two difficulties arise here. First, getting feedback from the NIDS (point 3 above) seems more problematic. In order for the attacker to determine whether an alarm has been generated or not, he would need to exploit an already compromised internal resource, such as for example an employee or device that provides him with this information. Alternatively, side channels may also be a source of valuable information (in particular, timing channels [39, 40]), for example if it takes a different amount of time to classify a normal and an anomalous request, and this can be remotely determined. The second difficulty has to do with the fact that during the attack Anagram receives a large amount of queries, many of which will be tagged as anomalous. As this might certainly raise some suspicions, the attacker could spread them over a much larger period of time.

### *4.2. Mask recovering algorithm*

We next describe an algorithm that recovers the secret mask applied by randomized Anagram. For reasons that will be clearer later, in some cases our attack could fail to locate exactly the borders between sets. Fortunately, such errors can only occur in the proximity of the borders and, therefore, the majority of the recovered mask is correct. Furthermore, the masks thus recovered are still extremely useful for an adversary to launch an evasion attack, as they point out which parts of the payload will be grouped together for analysis, even if there is some uncertainty about a few bytes at the beginning and end of each block.

For readability, in this section we present a high-level description of the attack. The pseudo-code of all involved procedures along with a detailed explanation can be found in Appendix A.

As described above, Anagram's masks are formed by concatenating runs of length at least 10 of natural numbers from the set $[0, K]$. As shown in Figure 3, our attack requires two inputs: (1) the maximum estimated size of the mask; and (2) the maximum estimated number $K$ of sets. The attack would be successful if both parameters are greater than or equal to the actual ones in the mask. However, these inputs have a direct influence on the execution time of the attack, in such a way that a more conservative (or resourceful) adversary could just use sufficiently high values to guarantee that the recovered mask is correct. Alternatively, it is possible to launch several attack instances, each one with a progressively higher value, until the result does not change.

The attack returns a vector with the estimated random mask, each position indicating the estimated set number. We will use the term *delimiter* to designate those mask positions where the mask changes from one set to another; in particular, if $m_i m_i \cdots m_i m_j m_j \cdots m_j$ are two consecutive blocks in the mask, with $m_i, m_j \in [0, K]$ and $m_i \neq m_j$, then the delimiter is the first $m_j$.

The algorithm is iterative. At each iteration, it identifies a new set $S_{current}$. The attack stops either when all the mask positions are filled with a set number, or when the maximum number of sets is reached. Each iteration of the algorithm is composed of two phases, which are explained in Sections 4.2.1 and 4.2.2,

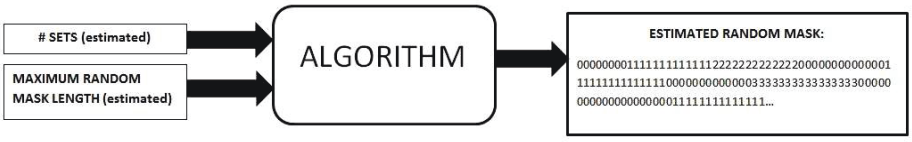



Figure 3: Input-Output view of the mask recovering attack.

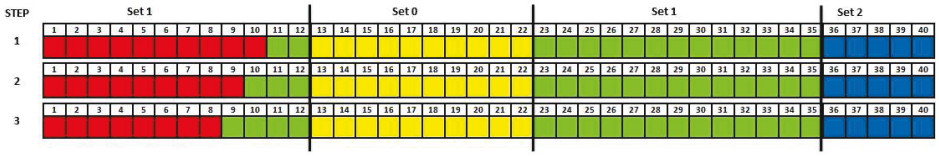



Figure 4: Phase I of the attack: obtaining an "almosts anomalous" payload.

respectively. Each phase uses as input an initial starting point ($I_s$), which corresponds to the first position of the delimiter in $S_{current}$. In the first iteration, $I_s$ is set to 1. In subsequent iterations, $I_s$ is the first delimiter found in the previous iteration such that no set has yet been assigned to it.

*4.2.1. Phase I*

The main goal of this phase is to construct a payload that is "almost anomalous." Such a payload is one that is classified as normal by Anagram, but such that if one single byte is replaced by a "malicious" one $\mu$ (which causes changes in $n$ consecutive $n$-grams), forces Anagram to classify it as anomalous[1]. The key observation here is that such a substitution will cause the payload to become anomalous if and only if $\mu$ is put in a position that belongs to the set with the maximum anomaly score ($S_{max}$ for convenience), and that is not close to the border with the next set so that most affected $n$-grams fall in $S_{max}$.

The payload is built up as represented in Figure 4. The algorithm starts with a normal payload and $I_s$ is set to 1. Then, $\mu$ is inserted into the 10 positions that follow $I_s$ (red cells in Figure 4). Next, the algorithm checks whether the payload becomes anomalous (i.e. whether $S_{current}$, which is set 1 in Figure 4, is $S_{max}$). If so, the algorithm removes one by one the $\mu$ bytes starting at the end (steps 2 and 3 in Figure 4), until the payload becomes normal again (step 3 in the Figure 4). The resulting payload is such that, while it is classified as normal, a single addition of $\mu$ in any of the positions associated with $S_{max}$ would cause it to become anomalous. Note that after Phase I, we guarantee that $S_{current} = S_{max}$.

[1]In our experiments, we have used as $\mu$ a byte with very low probability of occurrence in normal payloads, such as for example control characters (low ASCII numbers) or letters belonging to an alphabet different from the one used in the service being protected by the NIDS (e.g., ñ, accents, etc. if the system serves English web pages).

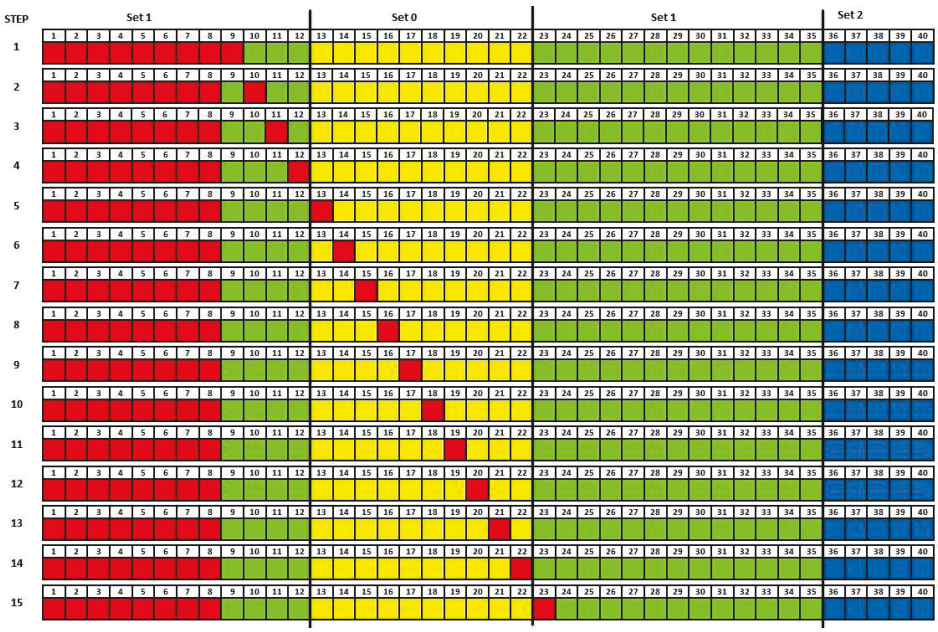



Figure 5: Phase II of the attack: $\mu$ is moved throughout the payload to detect changes in Anagram's ouput. Each change corresponds to a delimiter of the set being processed. In the figure, the algorithm would detect the delimiters of Set 1, located at positions 13, 23 and 36.

### *4.2.2. Phase II*

By using the payload obtained in Phase I, Phase II exploits the fact that the addition of a single $\mu$ into the sets of $S_{max}$ would cause the entire payload to become anomalous, whereas if it is inserted in any other set the payload remains normal. Phase I obtains payloads such that $S_{current}$ is $S_{max}$. Now Phase II moves one $\mu$ over the entire packet, as shown in Figure 5. Whenever such a moving $\mu$ is inserted within the limits of $S_{max}$, the payload becomes anomalous; otherwise it remains normal. This allows us to identify all those positions of set $S_{current}$ in the mask. In addition, any position where the output changes from normal to anomalous, or vice versa, is considered a delimiter between set $S_{current}$ and any other set. Thus, the next starting point $I_s$ will be the first delimiter belonging to a set that has not yet been assigned.

There are some border cases where the procedure described above may fail. These are next discussed, together with a simple but effective countermeasure consisting of running Phase II multiple times and carrying out a majority voting step.

### *4.2.3. Majority voting*

In the description of Phase II above it was assumed that the set obtained in Phase I, $S_{current}$, is $S_{max}$. Thus, $\mu$ is moved throughout the entire payload to detect where the output changes, therefore detecting the delimiters of the

current set. However, the algorithm fails if the positions close to the delimiters contain an already anomalous $n$-gram. The problem is that the payloads obtained in Phase I are "almost anomalous", meaning that one single $\mu$ within the limits of $S_{max}$ usually induces a change in the output. The anomaly score, as explained in Section 3.2.2, is obtained by dividing the number of unseen $n$-grams by the total number of $n$-grams. Consequently, the output changes when the number of unseen $n$-grams increases. However, during Phase II, if $\mu$ is inserted within an already unseen $n$-gram, then the number of unseen $n$-grams remains constant, the output does not change and, therefore, the delimiter is not detected. This situtation, which decreases the effectiveness of the algorithm, can also be exploited to evade Anagram, as we discuss in Section 5.4.

In order to increase the robustness of our attack, we introduce a majority voting scheme. Instead of simply recording the results for a single payload obtained in Phase I, we use several of them. The algorithm records all the positions indicated by each payload (votes) and, if some position has at least one half of the votes, then it is considered a delimiter. Even in those cases where it is unclear where the delimiter is, an analysis of the number of votes on each position will allow the adversary to estimate zones where the delimiters are supposed to be, which is enough to evade the system. We show this fact in Sections 5.2 and 5.4.

## 5. Experimental setup and results

We have implemented Anagram using the pseudo-code available in [19]. Both our attack and Anagram's implementation have been written in Java. Experiments have been run in a dual-core machine with 4GB of RAM. We have trained and tested Anagram using the same HTTP datasets used by McPAD [10], another application-layer anomaly detector, which are freely available[2]. A summary of the number of payloads and the partition into training, validation and test sets is given in Table 1. To generate the bad-content model, we use the web-based signatures of Snort [41], as done originally in Anagram[3]. Furthermore, to avoid inserting normal $n$-grams into the bad-content model, we filter out the data using the validation set from the McPAD dataset and a list of known words of the HTTP protocol.

We performed the experiments using 3 different $n$-gram sizes: $n = 5$, $n = 6$ and $n = 7$. Both in the original Anagram paper and in our experiments these values translate into the best detection quality. The experimental results presented next are grouped into three sections. In Section 5.1 we assess the performance of randomized Anagram as suggested in the original paper and discuss its limitations in terms of detection quality. Subsequently, in Sections

[2]See `http://roberto.perdisci.com/projects/mcpad`

[3]We do not use any virus signatures, as Anagram's authors do not provide information about what kind of virus database they used.

Table 1: Description of the dataset

| | Training | Validation | Test | Total |
|---|---|---|---|---|
| Normal payloads | 102157 | 1521 | 1050 | 104728 |
| Attack payloads | – | – | 1050 | 1050 |
| Total | 102157 | 1521 | 2100 | 105778 |

5.2 and 5.3 we report on the accuracy and efficiency (in terms of queries and CPU time) of the attack, respectively.

### *5.1. Detection accuracy*

In this first experiment, we assess the detection accuracy of randomized Anagram and compare it with the non-randomized version of the detector. Wang et al. [13] reported results on randomized Anagram using a binary mask (i.e., with just 2 sets). As our attack is designed to estimate random masks composed of any number of sets, we also explored the performance of randomized Anagram using a number of sets greater than 2, with different mask lengths. Specifically, we have experimented with 2, 3, 4, 5, 6 and 7 sets, and mask lengths of 128, 160, 200 and 256 bytes. In this section, we first present a ROC analysis of the randomized detectors. We next analyze the effect of introducing randomization on the anomaly score distribution, showing that the anomaly threshold in the case of randomized detection has to be increased in order to maintain a low false alarm rate.

#### *5.1.1. ROC analysis*

Figure 6 shows the ROC curves of the detectors using $n$-grams of size 5, 6 and 7. Each plot contains a ROC curve for different number of sets, which has been obtained by averaging the results of different mask lengths. These curves are similar to those originally presented by Wang et al. in [13]. It can be observed that for any value of $n$, as the number of sets increases, so it does the false alarm rate. However, all detectors achieve a 100% of detection rate with a false alarm rate lower than 1%. A false alarm rate greater than 1% has traditionally being considered unmanageable for a human operator at a large installation [42]. Accordingly, a typical design goal is to develop NIDS that can operate at points with a false alarm rate under 1%.

#### *5.1.2. Anomaly Score analysis*

Figure 7 shows the distribution of anomaly scores obtained during test using various $n$-gram sizes for attack-free packets (FREE, in red) and packets containing Polymorphic Blending Attack (PBA, in blue). Each plot shows the anomaly score in the $x$-axis and the number of payloads having such an anomaly score in the $y$-axis. Dotted lines represent the results of the randomized detector using a mask of 128 bytes with 2 sets, while solid lines show the results using a normal, non-randomized version of Anagram.

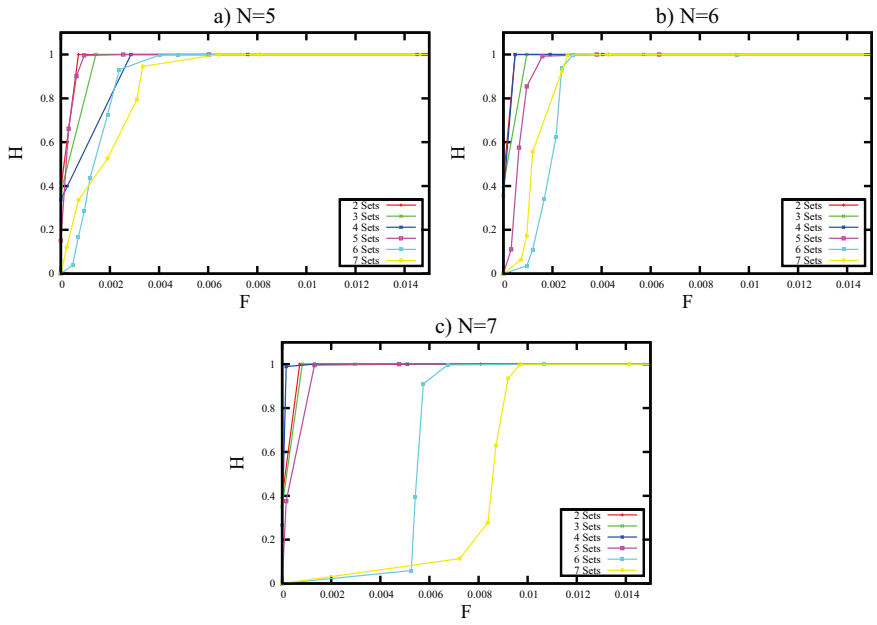



Figure 6: ROC curves obtained by randomized Anagram for different number of sets and different $n$-gram sizes.

As it can be observed, the randomized detector considerably increases the anomaly scores for both FREE and PBA packets. In our experience, this increment is even greater for attack-free packets. Although both distributions remain reasonably distinguishable, the detection threshold in the case of randomized testing is significantly higher. As we explain in Section 5.4, this property makes Anagram less robust against an adversary who has discovered the random mask.

### *5.2. Effectiveness of the attack*

Figure 8 shows an example of the execution of the attack using 3 sets and masks of 160 bytes. The figure is partitioned into three vertical blocks that should be viewed as concatenated. The actual mask content is shown in the first row (with the tag M). We run the attack using a maximum number of 8 sets and a maximum random mask length of 300 bytes. Each set is represented with a different color. The state of the estimated random mask after each iteration is shown in a different line. Thus, the second row corresponds to the initial state, where all the positions are set to -1 (in red). In the first iteration of the attack, the algorithm estimates the positions of the first set (0, in yellow). In the second iteration, the positions of set 2 are estimated (light green in the figure), while in the third step the algorithm finds the positions of set 3 (dark green). After step 3, there is a positive value in every position of the estimated random mask, so the algorithm stops and returns the estimated mask which, in this case, perfectly matches the original.

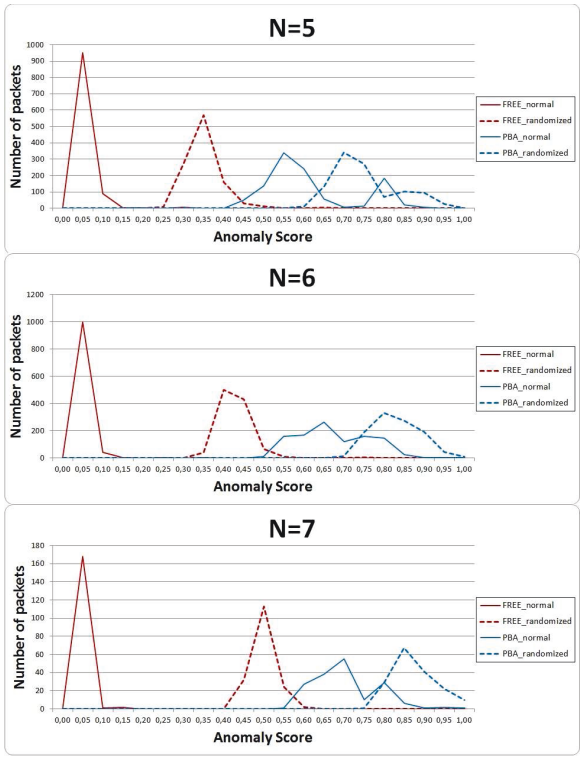



Figure 7: Distribution of the anomaly scores using 5-, 6- and 7-grams in non-randomized (plain lines) and randomized (dotted lines) Anagram.

In the case shown in Figure 8, the attack recovers exactly the random mask used. However, due to the situation explained in Section 4.2.3, in some cases the algorithm fails to correctly identify some sets. In order to evaluate such errors, we calculated the distance between the mask estimated by the attack and the actual one, measured as the number of incorrectly guessed sets divided by the total number of sets to normalize the results. We repeated each experiment 10 times with randomly generated masks for different Anagram configurations (i.e., varying the size of the $n$-grams, the mask size, and the number of sets in each mask) and computed the average distance between the recovered and the actual mask.

The number of packets to be used in the voting process depends on the desired accuracy. Figure 9 shows the error obtained when varying the number of voting packets for three different Anagram configurations. For the configurations used in this paper, it was experimentally determined that no significant error reduction is achieved with more than 30-40 votes. For example, in the case of random masks of 128 bytes with 2 sets, only 15 packets would be enough to reveal the mask perfectly. Accordingly, for a mask of 200 bytes and 4 sets, using

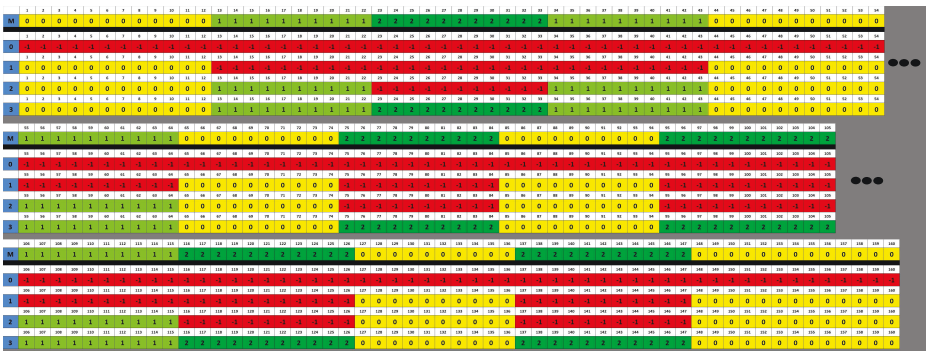


Figure 8: Evolution of the estimated random mask through consecutive iterations of the attack. The first line shows the actual mask with 3 different sets. The second line shows the initial state with all the positions set to -1, while each remaining line corresponds to an iteration of the algorithm.

20 packets would reveal it as well. Even in the case of a more complex configuration, such as the one with a random mask of 256 bytes and 7 sets, 40 packets correctly recover 82% of the mask (i.e. an error of 18%). As we discuss later, this percentage provides enough information to accomplish an evasion attack against Anagram.

In summary, the expected error depends both on the number of sets and the size of the random mask. Table 2 shows the experimental results for two different attack configurations using 10 and 40 votes, respectively. Several conclusions can be drawn:

- Binary masks are very easy to recover, no matter the size of the $n$-grams and, to an extent, the number of voting packets.
- The attack's probability of error increases with the number of sets and the mask length. The size of the $n$-grams seems to have no significant influence.
- Error decreases as the number of voting packets increases. For example, while in the case of 10 packets the average error falls between 0.08 and 0.68, it is reduced to less than 0.07 when 40 packets are used.

An interesting property of our mask recovery process is that, even if it does not estimate the mask exactly, an adversary can figure out approximately where the delimiters of the mask are by just looking at the votes, as shown in Figure 10. This plot represents the number of votes obtained for a random mask of length 128 and 3 sets using 7-grams. The $x$-axis shows the mask positions, while the $y$-axis shows the number of votes (i.e., packets indicating that there is a delimiter in this position). Take for example position 74, which is an actual delimiter. The number of votes after iterations 2 and 3 are not enough, as the final count does not reach half of the votes, so the majority voting scheme does not determine that there is a delimiter there. As a consecuence, the algorithm fails and the

Table 2: Average distance between estimated and actual masks. Each row corresponds to a different number of sets ($K$), while each column determines the mask length and the size $N$ of the $n$-grams.

| | **10 voting packets** | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | N=5 | | | | N=6 | | | | N=7 | | | | Average |
| $K$ | 128 | 160 | 200 | 256 | 128 | 160 | 200 | 256 | 128 | 160 | 200 | 256 | |
| 2 | 0.18 | 0.02 | 0.02 | 0.02 | 0.05 | 0.02 | 0.00 | 0.02 | 0.26 | 0.02 | 0.20 | 0.13 | 0.08 |
| 3 | 0.42 | 0.49 | 0.54 | 0.58 | 0.40 | 0.48 | 0.53 | 0.55 | 0.54 | 0.56 | 0.47 | 0.50 | 0.51 |
| 4 | 0.39 | 0.51 | 0.62 | 0.70 | 0.37 | 0.57 | 0.60 | 0.65 | 0.42 | 0.56 | 0.57 | 0.70 | 0.56 |
| 5 | 0.50 | 0.59 | 0.64 | 0.71 | 0.53 | 0.70 | 0.63 | 0.69 | 0.48 | 0.60 | 0.62 | 0.70 | 0.62 |
| 6 | 0.53 | 0.64 | 0.63 | 0.75 | 0.56 | 0.68 | 0.64 | 0.71 | 0.58 | 0.65 | 0.69 | 0.65 | 0.64 |
| 7 | 0.61 | 0.60 | 0.75 | 0.75 | 0.55 | 0.65 | 0.77 | 0.76 | 0.65 | 0.64 | 0.71 | 0.75 | 0.68 |

| | **40 voting packets** | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | N=5 | | | | N=6 | | | | N=7 | | | | Average |
| $K$ | 128 | 160 | 200 | 256 | 128 | 160 | 200 | 256 | 128 | 160 | 200 | 256 | |
| 2 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 3 | 0.00 | 0.00 | 0.00 | 0.03 | 0.01 | 0.00 | 0.00 | 0.00 | 0.02 | 0.00 | 0.00 | 0.00 | 0.00 |
| 4 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.02 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 5 | 0.00 | 0.06 | 0.06 | 0.12 | 0.05 | 0.00 | 0.00 | 0.04 | 0.00 | 0.00 | 0.11 | 0.00 | 0.04 |
| 6 | 0.00 | 0.00 | 0.00 | 0.02 | 0.08 | 0.00 | 0.03 | 0.16 | 0.16 | 0.00 | 0.04 | 0.42 | 0.08 |
| 7 | 0.03 | 0.00 | 0.00 | 0.05 | 0.11 | 0.03 | 0.03 | 0.11 | 0.16 | 0.09 | 0.12 | 0.18 | 0.07 |

remaining sets may or may not be properly estimated. This limitation can be detected either by a graphical analysis of the voting results or, alternatively, by adjusting the number of votes required to achieve majority. This would allow the adversary to determine which zones of the payload are very likely to contain no delimiters. Thus, even if the exact mask is not recovered, this information may suffice to perform an evasion attack, as we later illustrate in Section 5.4.

### *5.3. Efficiency of the attack*

In the previous section we have shown that the proposed attack succeeds in recovering the random mask or, at least, gives enough information about its structure. However, when trying to evade any security system, it is critical to do so spending as few computational resources as possible. Figure 11 shows the number of queries to Anagram and the CPU time (in seconds) required by the algorithm for different number of sets. In general, the time required by the attack directly depends on the number of queries made to Anagram. In turn, the number of queries strongly depends on Phase I of the attack, where a "nearly-anomalous" payload is obtained, hence that the data range (size of each box) is relatively large. For example, using 40 voting packets, the attack requires between 40 seconds (for binary masks) and 120 seconds (for masks composed of 7 sets), with the average number of queries ranging between 100 and 6000.

### *5.4. Exploiting randomization to evade detection*

Figure 7 shows that the anomaly score using randomized testing is typically larger than using normal testing, both for attack-free and PBA payloads. This happens because those payload bytes that are placed in positions around a mask

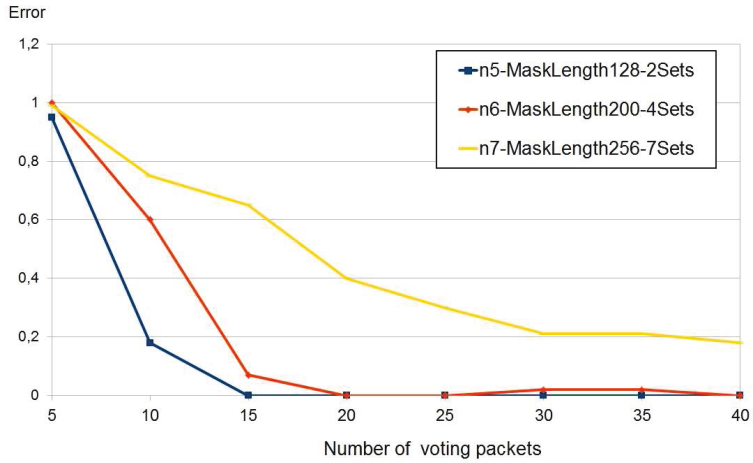



Figure 9: Decrease of the average distance between recovered and actual mask as the number of voting packets increase.

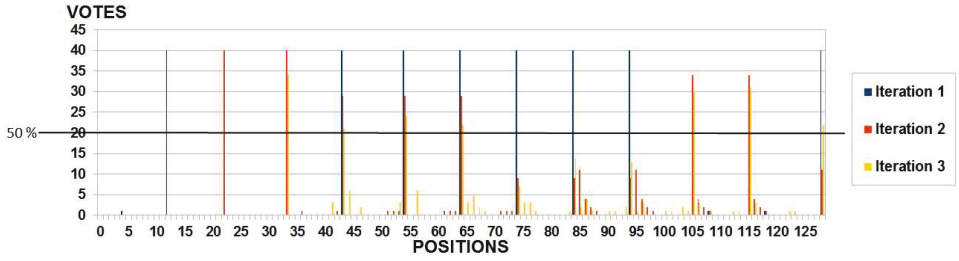
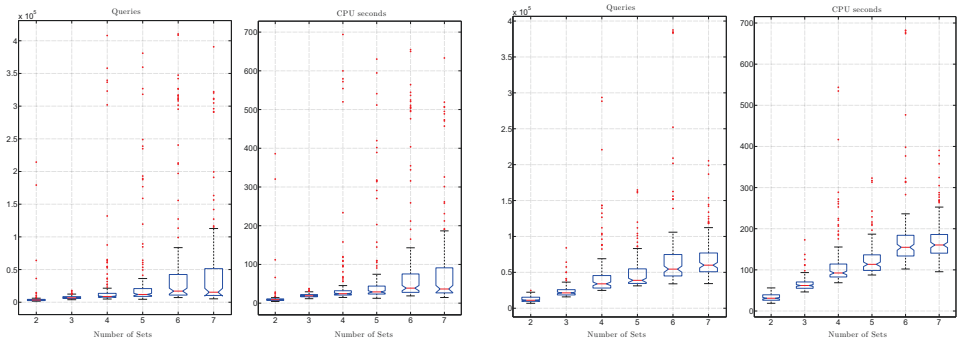



Figure 10: Votes obtained when estimating the mask of 3 sets and 128 bytes length

delimiter are partitioned and concatenated with other chunks of data. In this process, several unseen $n$-grams may appear, even with bytes of normal data. Therefore, in order to achieve a good detection rate while minimizing the false positive rate, the anomaly threshold must be increased.

If an adversary is able to obtain the random mask being applied, then he can use the randomized testing process to evade the system. As mentioned above, when using randomization the threshold should be increased, thus tolerating the presence of more malicious bytes in the payload. Such malicious bytes are supposed to be in the positions of the mask delimiters, so an adversary can generate a packet where the malicious content is placed exactly in these positions, padding the remaining parts of the payload with normal bytes. Moreover, if the adversary suspects that parts of the malicious payload appear in the bad-content model of Anagram, then he can distribute this content around the delimiters too, in such a way that it will be split and will no longer match this bad-content model.




(a) 10 voting packets (b) 40 voting packets

Figure 11: Queries and CPU time required to estimate the random masks with different number of sets.

Figure 12 exemplifies this idea with a simple instance of an XSS attack using javascript. The attack actually just launches a pop-up window in the client side. The first payload only contains the code needed to perform the attack. In the second payload, padding is added to reduce the percentage of unseen $n$-grams. Assume that "image/" is a string that appears in the normal model. In such a case, padding is inserted as comments of the language (after the characters // and between /* and */).

The third payload in Figure 12 refers to the same attack, but is especially crafted to evade detection by using the estimated mask. We assume that the string "<script>" is in the bad-content model, as it is a frequent word in XSS attacks. Therefore, when preparing the payload, this word should be placed around some delimiter in order to have it split. Afterwards, the attacker places the desired amount of padding bytes in proper places to benefit from the randomization process. Figure 13 shows an example of such a payload prepared for a random mask of 2 sets. It can be seen that when using randomized testing, the word "<script>" will be divided into the 5-gram "<scr*/" and tested as part of the set 1, and the 5-gram "ipt>/", tested as part of the set 0. Table 3 shows the anomaly scores obtained for each case. For the original payload (first row), both anomaly scores (for normal and randomized testing) are very large, which means that the payload is undoubtedly considered as an anomaly. For the modified payload with normal padding (second row) the anomaly score is 0.45, exceeding the threshold established and therefore being considered anomalous again. However, when using randomized testing and preparing the rogue payload with padding inserted in the proper positions (third row), the anomaly score obtained is even lower than the one of the normal test. Moreover, as discussed in Section 5.1, the anomaly threshold is higher when using randomization than when using normal testing. In fact, in this case the anomaly threshold is 0.55, so the attack payload will be classified as normal, and thus evasion would succeed. This example illustrates that the randomized testing is a double-edged sword, since if the adversary is able to guess the mask, then he can use it against

1) Original attack payload

```
<script>
alert('attack')
</script>
```

2) Attack payload with padding (normal test)

```
image/
<script>
/*image/*/ alert(\'attack;\');
//image/image/image/image/image/image/image/image/image/image/image/image/image/image/imag
</script>
```

3) Attack payload with prepared padding (randomized test)

```
image/
<script>
/*image/*/ alert(\'attack;\');
//image/imageimage/imag/image/imagee/image/imag/image/imae/image/image/image/image/image/
</script>
```

Figure 12: Examples of attack payloads used to evade Anagram. Payload 1 is the original without padding. Payload 2 adds normal padding. Payload 3 is crafted to evade Anagram once the random mask has been estimated.

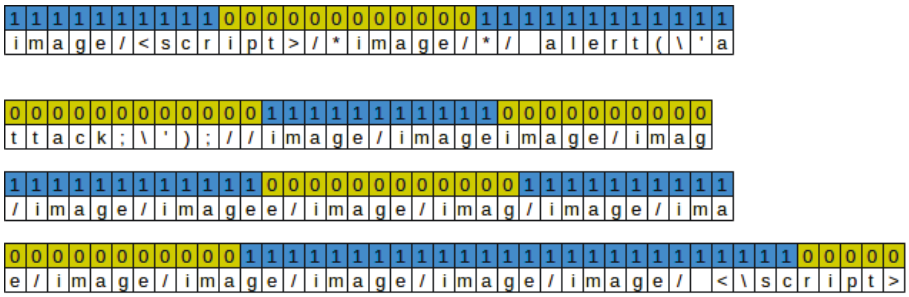



Figure 13: Example of how an adversary can set up an attack using the estimated random mask to evade Anagram.

the detection system.

Table 3: Anomaly scores obtained with the original payload, the payload with normal padding, and the payload prepared to evade the system using the estimated random mask.

| | Normal testing | Random testing |
|---|---|---|
| 1) Original payload | 1.50 | 2.00 |
| 2) Payload with padding | 0.45 | 0.53 |
| 3) Payload with prepared padding | 0.60 | 0.38 |

## 6. Conclusions

In this paper, we have analyzed the strength of randomized Anagram against mask recovery attacks. Even though the use of randomization certainly makes evasion harder, we have shown that an adversary who manages to find out such a mask could actually take advantage of the randomized detection process to evade Anagram, thus turning a security measure into an undesirable feature. We have proposed and evaluated a procedure to recover the secret mask by querying Anagram with carefully constructed payloads and observing the results. Our attack is quite efficient in terms of the number of queries employed, requiring no more than 2 minutes to recover the mask in the worst scenario for the range of the suggested parameters. As discussed above, we do not make any claims about the feasibility of the proposed attack in real-world scenarios, as this strongly depends on the adversary having the ability to interact with Anagram in the ways detailed in Section 4.1.

A possible countermeasure to the proposed attack is to randomize the choosing of random mask itself. Thus, each analyzed packet should be tested against a different random mask, possibly with different parameters too. While this would certainly stop our attacks from being effective, we have not assessed the potential impact of such a double randomization from the detection point of view.

[1] K. A. Scarfone, P. M. Mell, SP 800-94. Guide to Intrusion Detection and Prevention Systems (IDPS), Technical Report, Gaithersburg, MD, United States, 2007.

[2] I. Corona, G. Giacinto, F. Roli, Adversarial attacks against intrusion detection systems: Taxonomy, solutions and open issues, Information Sciences 239 (2013) 201–225.

[3] B. Hu, Y. Shen, Machine learning based network traffic classification: a survey, Journal of Information and Computational Science 9 (2012) 3161–3170. Cited By (since 1996)0.

[4] S. Pastrana, A. Mitrokotsa, A. Orfila, P. Peris-Lopez, Evaluation of classification algorithms for intrusion detection in MANETs, Knowledge-Based Systems 36 (2012) 217–225.

[5] J. Song, H. Takakura, Y. Okabe, K. Nakao, Toward a more practical unsupervised anomaly detection system, Information Sciences: an International Journal 231 (2013) 4–14.

[6] K. Satpute, S. Agrawal, J. Agrawal, S. Sharma, A survey on anomaly detection in network intrusion detection system using particle swarm optimization based machine learning techniques, Advances in Intelligent Systems and Computing 199 AISC (2013) 441–452.

[7] P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, W. Lee, Polymorphic blending attacks, in: Proceedings of the 15th USENIX Security Symposium, USENIX, Vancouver, BC, Canada, 2006, pp. 241–256.

[8] O. Kolesnikov, W. Lee, Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic, Technical Report, Georgia Institute of Technology, 2005.

[9] K. Wang, S. Stolfo, Anomalous payload-based network intrusion detection, in: E. Jonsson, A. Valdes, M. Almgren (Eds.), Recent Advances in Intrusion Detection, volume 3224 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2004, pp. 203–222.

[10] R. Perdisci, D. Ariu, P. Fogla, G. Giacinto, W. Lee, McPAD: A Multiple Classifier System for Accurate Payload-based Anomaly Detection, Computer Networks 53 (2009) 864–881.

[11] B. Biggio, G. Fumera, F. Roli, Multiple classifier systems for robust classifier design in adversarial environments, International Journal of Machine Learning and Cybernetics 1 (2010) 27–41.

[12] S. Mrdovic, B. Drazenovic, Kids: keyed intrusion detection system, in: Proceedings of the 7th international conference on Detection of intrusions and malware, and vulnerability assessment, DIMVA'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 173–182. URL: `http://dl.acm.org/citation.cfm?id=1884848.1884862`.

[13] K. Wang, J. J. Parekh, S. J. Stolfo, Anagram: A content anomaly detector resistant to mimicry attack, in: Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection, volume 4219 of *Lecture Notes in Computer Science*, Springer, Hamburg, Germany, 2006, pp. 226–248.

[14] S. Lee, G. Kim, S. Kim, Self-adaptive and dynamic clustering for online anomaly detection, Expert Systems with Applications 38 (2011) 14891–14898.

[15] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, K.-Y. Tung, Intrusion detection system: A comprehensive review, Journal of Network and Computer Applications 36 (2013) 16–24.

[16] D. Ariu, R. Tronci, G. Giacinto, Hmmpayl: An intrusion detection system based on hidden markov models, Computers and Security 30 (2011) 221–241.

[17] A. Aziz, M. Salama, A. ella Hassanien, S. El-Ola Hanafi, Detectors generation using genetic algorithm for a negative selection inspired anomaly network intrusion detection system, in: 2012 Federated Conference on Computer Science and Information Systems (FedCSIS), 2012, pp. 597–602.

[18] G. Kumar, K. Kumar, M. Sachdeva, The use of artificial intelligence based techniques for intrusion detection: a review, Artificial Intelligence Review 34 (2010) 369–387.

[19] K. Wang, Network payload-based anomaly detection and content-based alert correlation, Ph.D. thesis, New York, NY, USA, 2007.

[20] D. Hadziosmanovic, L. Simionato, D. Bolzoni, E. Zambon, S. Etalle, N-gram against the machine: On the feasibility of the n-gram network analysis for binary protocols., in: Recent Advances in Intrusion Detection, volume 7462 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 354–373.

[21] R. Sommer, V. Paxson, Outside the closed world: On using machine learning for network intrusion detection, in: Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 305–316. URL: `http://dx.doi.org/10.1109/SP.2010.25`. doi:`10.1109/SP.2010.25`.

[22] Y. Song, M. E. Locasto, A. Stavrou, A. D. Keromytis, S. J. Stolfo, On the infeasibility of modeling polymorphic shellcode, in: Proceedings of the 14th ACM conference on Computer and communications security, CCS '07, ACM, New York, NY, USA, 2007, pp. 541–551. URL: `http://doi.acm.org/10.1145/1315245.1315312`. doi:`10.1145/1315245.1315312`.

[23] T. H. Ptacek, T. N. Newsham, Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection, Technical Report, Secure Networks, Inc., Syracuse, NY, USA, 1998.

[24] U. Shankar, Active mapping: Resisting nids evasion without altering traffic, in: Proceedings of the 2002 IEEE Symposium on Security and Privacy, Oakland, California, USA, 2003, pp. 44–61.

[25] M. Vutukuru, H. Balakrishnan, V. Paxson, Efficient and robust tcp stream normalization, in: Proceedings of the 2008 IEEE Symposium onon Security and Privacy, Oakland, California, USA, 2008, pp. 96–110.

[26] D. Mutz, G. Vigna, R. Kemmerer, An experience developing an ids stimulator for the black-box testing of network intrusion detection systems, in: Proceedings of the 19th IEEE Annual Computer Security Applications Conference, 2003, pp. 374–383.

[27] G. Vigna, W. Robertson, D. Balzarotti, Testing network-based intrusion detection signatures using mutant exploits, in: Proceedings of the 11th ACM Conference on Computer and Communications Security, ACM, Washington, DC, USA, 2004, p. 21.

[28] R. Perdisci, D. Dagon, W. Lee, P. Fogla, M. Sharif, Misleading worm signature generators using deliberate noise injection, in: Proceedings of the 2006 IEEE Symposium on Security and Privacy, IEEE, 2006, pp. 17–31.

[29] S. P. Chung, A. K. Mok, Advanced allergy attacks: Does a corpus really help?, in: Recent Advances in Intrusion Detection, Springer, 2007, pp. 236–255.

[30] M. Kloft, P. Laskov, Online anomaly detection under adversarial impact, Journal of Machine Learning Research - Proceedings Track 9 (2010) 405–412.

[31] M. Barreno, B. Nelson, R. Sears, A. D. Joseph, J. Tygar, Can machine learning be secure?, in: Proceedings of the 2006 ACM Symposium on Information, computer and communications security, ACM, 2006, pp. 16–25.

[32] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, J. D. Tygar, Adversarial machine learning, in: Proceedings of the 4th ACM workshop on Security and artificial intelligence, AISec '11, ACM, New York, NY, USA, 2011, pp. 43–58. URL: http://doi.acm.org/10.1145/2046684.2046692. doi:10.1145/2046684.2046692.

[33] B. Biggio, G. Fumera, F. Roli, Security evaluation of pattern classifiers under attack, IEEE Transactions on Knowledge and Data Engineering 99 (2013) 1.

[34] S. Pastrana, A. Orfila, A. Ribagorda, A functional framework to evade network ids, in: Proceedings of the 44th IEEE Hawaii International International Conference on Systems Sciences, Koloa, Hawaii, USA, 2011, pp. 1–10.

[35] P. Fogla, W. Lee, Evading network anomaly detection systems: Formal reasoning and practical techniques, in: Proceedings of the 13th ACM Conference on Computer and Communications Security, ACM, Alexandria, VA, USA, 2006, pp. 59–68.

[36] B. H. Bloom, Space/time trade-offs in hash coding with allowable errors, Communications of the ACM 13 (1970) 422–426.

[37] M. Barreno, B. Nelson, A. D. Joseph, J. Tygar, The security of machine learning, Machine Learning 81 (2010) 121–148.

[38] D. Lowd, C. Meek, Adversarial learning, in: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining, ACM, 2005, pp. 641–647.

[39] S. Chen, R. Wang, X. Wang, K. Zhang, Side-channel leaks in web applications: A reality today, a challenge tomorrow, in: Security and Privacy (SP), 2010 IEEE Symposium on, IEEE, 2010, pp. 191–206.

[40] D. Brumley, D. Boneh, Remote timing attacks are practical, Computer Networks 48 (2005) 701–716.

[41] M. Roesch, Snort: Lightweight intrusion detection for networks., in: Proceedings of the 13th Systems Administration Conference, USENIX, Seattle, WA, USA, 1999, pp. 229–238.

[42] T. Champion, R. Durst, Air force intrusion detection system evaluation environment., in: Recent Advances in Intrusion Detection (RAID), 1999.

## Appendix A. Detailed Description of the Attack

The attack to recover the secret mask is divided into several functions. In order to limit the execution of the algorithm, it takes as inputs two estimated thresholds: the number of sets and the maximum random mask length (see Figure 3). We next provide a pseudocode description of the attack divided into 4 algorithms, namely Algorithm 1, 2, 3 and 4.

As shown in Algorithm 1, the attack starts by initializing the mask with the value '-1'. The value '-1' in the position $I$ means that the algorithm has not yet obtained the set corresponding to $I$. The algorithm also initializes an empty list of the positions that are first delimiters. These two structures are global to all the processes. At each iteration of the algorithm's main loop, the function *findSet* is called with the starting point $I_s$ and the current set $S_{current}$. This starting point is the position from which the algorithm will search delimiters of the set starting in this position ($S_{current}$). In the first step, the algorithm starts at 0 and obtains the delimiters of the set 0 of the mask. Once obtained, it will proceed similarly but starting from a new point (line 10 in Algorithm 1), specifically the next delimiter whose set has not been processed yet (i.e., a delimiter whose next position in the mask is still '-1').

The function *findSet*, shown in Algorithm 2, receives the starting position $I_s$ and the current set $S_{current}$. This is the core function of the attack. First, it selects a payload $P$ which is considered "normal" by calling the function *anagramTest* (lines 4-9 in Algorithm 2). Next, it looks for a "nearly-anomalous" payload $P^*$ that is used in the following steps of the algorithm. It does so by calling the function *getValidPayload*, which is shown in Algorithm 4 and explained in Figure 4. The malicious byte $\mu$ is inserted into the 10 positions immediately next to the starting point of payload $P$ (lines 2-5 in Algorithm 4). If this modification causes the payload to become anomalous (line 7 in Algorithm 4), then it removes the byte $\mu$ backwards, one position at a time, until the packet becomes normal again. This process will only work with payloads $P$ that are close enough to become anomalous. As we can only use 10 positions following the starting point $I_s$ (by definition these are the only positions that belong to

the same set with certainty), we need payloads classified as normal that, with the addition of just a few previously unseen bytes, become anomalous.

Due to the behavior of the randomized test of Anagram, $P^*$ (function *findSet* showed in Algorithm 2) will become anomalous only when $\mu$ is inserted into chunks of data that are mapped into the set $S_{current}$. The key point here is that when $\mu$ is inserted in the region of a set different from $S_{current}$, the *anagramTest* will still output "normal", as this $\mu$ is no longer considered within the bytes of the set $S$. Therefore, by sliding the malicious byte through the payload (lines 17-28 in Algorithm 2), as shown in the Figure 5, and looking where the output of *anagramTest* changes, we can estimate where the delimiters of the set $S_{current}$ are. Regarding Figure 5, in the steps 1, 2, 3 and 4, *anagramTest* may output "anomaly" because the malicious byte is still inserted in the *Set 1*, whereas steps 5 to 14 may output "normal". However, in the step 15, as the malicious byte is inserted again into the *Set 1*, it may output again "anomaly". In order to optimize the process and to avoid unnecessary queries to Anagram, this step is skipped (line 18 in Algorithm 2) if the position where $\mu$ is going to be inserted has a value different from '-1' in the estimated random mask. We repeat this process for several payloads and record the delimiters indicated by each of them, which we call VOTES (line 24 in Algorithm 2). The final step of the algorithm is to process all votes using the function *processVotes* (line 31 in Algorithm 2).

The function *processVotes*, shown in Algorithm 3, processes the votes obtained by all payloads. We consider that a position is a delimiter of $S_{current}$ if it is supported by at least half of the votes (line 4 in Algorithm 3). If so, we add this position to the final list of delimiters of $S_{current}$: $D_1, D_2, ..., D_d$. The algorithm also saves the list of possible next starting points ($FIRST\_DELIMITERS$). Then, the estimated mask is updated by setting the set $S_{current}$ to the positions between the consecutive delimiters $D\_a$ and $D\_b$ (lines 15-17 of 3). Finally, the function obtains and returns the next starting delimiter whose set has not been already obtained (lines 22-26 in Algorithm 3). This $NEXT\_I_s$ delimiter will be the next starting point ($I_s$) for the algorithm (line 5 in Algorithm 1).

**Algorithm 1** $FindMask$

**Input:** Number of estimated Sets NS and maximum estimated random length $MAX\_LENGTH$

**Output:** Estimated $MASK$.

1: $MASK = \{-1, ..., -1\}$
2: $FIRST\_POSITIONS = \emptyset$
3: $I_s \leftarrow 0$
4: **for** $S = 0 \rightarrow NS$ **do**
5: $\quad \{NEXT\_I_s\} \leftarrow findSet\,(I_s, S)$
6: $\quad$ **if** $NEXT\_I_s = NULL$ **then**
7: $\quad\quad$ **return** $MASK$
8: $\quad$ **end if**
9: $\quad I_s \leftarrow NEXT\_I_s$
10: $\quad S \leftarrow S + 1$
11: **end for**
12: **return** $MASK$

**Algorithm 2** $findSet$

**Input:** Initial position $I_s$ and current set $S$
**Output:** Next starting positions $NEXT\_I_s$.

1: $VALID = 0$
2: $VOTES = \{0, ..., 0\}$
3: **while** $VALID < NUMPAYLOADS$ **do**
4: $P \leftarrow getPayloadFromPool\,()$
5: $OUTPUT \leftarrow anagramTest\,(P)$
6: **while** $OUTPUT \neq' NORMAL'$ **do**
7: $P \leftarrow getPayloadFromPool\,()$
8: $OUTPUT \leftarrow anagramTest\,(P)$
9: **end while**
10: $P^* \leftarrow getValidPayload\,(P, I_s)$
11: **if** $P^* \neq NULL$ **then**
12: $VALID \leftarrow VALID + 1$
13: $POSITION \leftarrow lastIndexOf\,(\mu, P^*) + 1$
14: $LOOKER \leftarrow P^*$
15: $LOOKER[POSITION] = \mu$
16: $PREVIOUS \leftarrow anagramTest\,(LOOKER)$
17: **while** $POSITION < MASK.LENGTH$ **do**
18: **if** $MASK[POSITION] < 0$ **then**
19: $POSITION \leftarrow POSITION + 1$
20: $LOOKER \leftarrow P^*$
21: $LOOKER[POSITION] = \mu$
22: $OUTPUT \leftarrow anagramTest\,(LOOKER)$
23: **if** $OUTPUT \neq PREVIOUS$ **then**
24: $VOTES[POSITION] \leftarrow VOTES[POSITION] + 1$
25: **end if**
26: **end if**
27: $PREVIOUS \leftarrow OUTPUT$
28: **end while**
29: **end if**
30: **end while**
31: **return** $processVotes\,(VOTES, VALID, S)$

**Algorithm 3** *processVotes*

**Input:** An array of votes $VOTES$, the number of valid packets $VALID$, the current set $S$, the current starting position $I_s$

**Output:** Next starting delimiter $NEXT\_I_s$.

```
1: DELIMITERS ← {I_s}
2: END_DELIMITER ←' TRUE'
3: for I = 0 → VOTES.LENGTH do
4:    if VOTES[I] >= VALID/2 then
5:       DELIMITERS.add(I)
6:       if END_DELIMITER then
7:          END_DELIMITER ←' FALSE'
8:          FIRST_DELIMITERS.add(I)
9:       else
10:         END_DELIMITER ←' TRUE'
11:      end if
12:   end if
13: end for
14: for I = 0 → DELIMITERS.LENGTH − 1 do
15:   for J = DELIMITERS[I] → DELIMITERS[I + 1] do
16:      MASK[J] = S
17:   end for
18:   I ← I + 2
19: end for
20: sort(FIRST_DELIMITERS)
21: NEXT_I_s ← FIRST_DELIMITER[0]
22: while MASK[NEXT_I_s + 1] > 0 AND NEXT_I_s ≠ NULL do
23:   FIRST_DELIMITERS.remove(0)
24:   NEXT_I_s ← FIRST_DELIMITER[0]
25: end while
26: return NEXT_I_s
```

**Algorithm 4** *getValidPayload*

**Input:** A payload $P$ and the starting position $I_s$.
**Output:** A payload $P^*$ or "$NULL$".

1: $P^* \leftarrow P$
2: **for** $i = I_s \rightarrow I_s + 10$ **do**
3: $\quad P^*[i] = \mu$
4: $\quad i \leftarrow i + 1$
5: **end for**
6: $output \leftarrow anagramTest\,(P)$
7: **if** $output =' ANOMALY'$ **then**
8: $\quad$ **for** $i = I_s + 10 \rightarrow I_s$ **do**
9: $\quad\quad P^*[i] = P[i]$
10: $\quad\quad output \leftarrow anagramTest\,(P^*)$
11: $\quad\quad$ **if** $output =' NORMAL'$ **then**
12: $\quad\quad\quad$ **return** $P^*$
13: $\quad\quad$ **end if**
14: $\quad\quad i \leftarrow i - 1$
15: $\quad$ **end for**
16: **end if**
17: **return** $NULL$